

COPY OF PAPERS
ORIGINALLY FILED



IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Re application of: A. Koseki et al.

Date: June 21, 2002

Serial No.: 10/063,958

Docket No.: JP920010018US1

Filed: May 29, 2002

Group Art Unit: 2122

For: **COMPILING METHOD AND STORAGE
MEDIUM THEREFOR**

Assistant Commissioner for Patents
Washington, D.C. 20231

SUBMISSION OF PRIORITY DOCUMENT

Sir:

Enclosed herewith is a certified copy of Japanese Application No. 2001-161364
filed May 29, 2001, in support of applicant's claim to priority under 35 U.S.C. 119.

Respectfully submitted,

Derek S. Jennings
Reg. Patent Agent/Patent Engineer
Reg. No. 41,473
(914) 945-2144

IBM CORPORATION
Intellectual Property Law Dept.
P. O. Box 218
Yorktown Heights, N. Y. 10598



日 本 国 特 許 庁
JAPAN PATENT OFFICE

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office

出 願 年 月 日

Date of Application:

2001年 5月29日

出 願 番 号

Application Number:

特願2001-161364

出 願 人

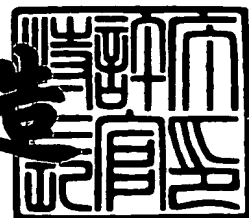
Applicant(s):

インターナショナル・ビジネス・マシーンズ・コーポレーション

2001年 7月27日

特 許 庁 長 官
Commissioner,
Japan Patent Office

及 川 耕 造



出証番号 出証特2001-3066762

【書類名】 特許願

【整理番号】 JP9010018

【提出日】 平成13年 5月29日

【あて先】 特許庁長官 殿

【国際特許分類】 G06F 5/06

【発明者】

【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 古関 聡

【発明者】

【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 竹内 幹雄

【発明者】

【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 小松 秀昭

【特許出願人】

【識別番号】 390009531

【氏名又は名称】 インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

【識別番号】 100086243

【弁理士】

【氏名又は名称】 坂口 博

【代理人】

【識別番号】 100091568

【弁理士】

【氏名又は名称】 市位 嘉宏

【代理人】

【識別番号】 100106699

【弁理士】

【氏名又は名称】 渡部 弘道

【復代理人】

【識別番号】 100104880

【弁理士】

【氏名又は名称】 古部 次郎

【選任した復代理人】

【識別番号】 100100077

【弁理士】

【氏名又は名称】 大場 充

【手数料の表示】

【予納台帳番号】 081504

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【包括委任状番号】 9706050

【包括委任状番号】 9704733

【包括委任状番号】 0004480

【プルーフの要否】 要

【書類名】 明細書

【発明の名称】 コンパイル方法、コード生成方法、スタックレジスタ使用方法、コンパイラ、これらを実現するプログラム及び記憶媒体

【特許請求の範囲】

【請求項 1】 ソースコードで記述されたプログラムをオブジェクトコードに変換するコンパイル方法において、

前記プログラムに対してレジスタを割り付けるステップと、

レジスタ割付の結果に基づいてオブジェクトコードを生成するステップとを含み、

前記レジスタを割り付けるステップは、

前記プログラム中の命令に論理レジスタを割り当てるステップと、

前記プログラムにおけるプロシージャの呼び出し時に生存している物理レジスタがレジスタスタックの底から順に割り付けられるように、前記論理レジスタと前記物理レジスタとの間のマッピングを行うステップと、を含むことを特徴とするコンパイル方法。

【請求項 2】 前記マッピングを行うステップは、生存区間が重なるプロシージャ呼び出しが多い前記論理レジスタほど優先させて前記物理レジスタに割り付けるステップを含むことを特徴とする請求項 1 に記載のコンパイル方法。

【請求項 3】 前記マッピングを行うステップは、生存区間が重なるプロシージャ呼び出し時点において、同時に生きている論理レジスタの数が少ないほど、当該論理レジスタを優先させて前記物理レジスタに割り付けるステップを含むことを特徴とする請求項 1 に記載のコンパイル方法。

【請求項 4】 コンピュータを制御するプログラムのコードを生成するコード生成方法において、

レジスタが所定の命令に割り付けられているかどうかを確認しながらコード生成を行い、

プロシージャ呼び出し時に、計算ユニット資源に空きがある限りにおいて、レジスタスタック上で所定の命令に割り付けられているレジスタの内容を、命令に割り付けられておらず当該レジスタよりもレジスタスタックの底側に位置するレ

ジスタにコピーすることを特徴とするコード生成方法。

【請求項5】 レジスタスタックを使用するアーキテクチャを持った処理装置におけるプログラム実行時のスタックレジスタ使用方法において、

所定のプロシージャにおいて他のプロシージャが呼び出された場合に、当該所定のプロシージャの実行のために割り付けられたレジスタのうちで当該他のプロシージャの呼び出し時に生存しているレジスタを割り付け直して当該他のプロシージャを呼び出すステップと、

前記他のプロシージャからの復帰が行われた場合に、レジスタイメージを割り付け直す前の状態に戻すステップとを含むことを特徴とするスタックレジスタ使用方法。

【請求項6】 前記レジスタを割り付け直して前記他のプロシージャを呼び出すステップは、

前記他のプロシージャの呼び出し時に生存しているレジスタを、レジスタスタックの底から順にソートして割り付け直すステップを含むことを特徴とする請求項5に記載のスタックレジスタ使用方法。

【請求項7】 レジスタスタックを使用するアーキテクチャを持った処理装置におけるプログラム実行時のスタックレジスタ使用方法において、

プロシージャの呼び出しを行うたびに、当該プロシージャの呼び出し時にスタックレジスタ上に生存しているレジスタをパッキングして割り付け、

前記プロシージャを実行した後に、レジスタイメージをパッキング前の状態に戻すことを特徴とするスタックレジスタ使用方法。

【請求項8】 プログラミング言語で記述されたプログラムのソースコードを機械語コードに変換するコンパイラにおいて、

前記プログラム中の命令に対してレジスタを割り付けるレジスタアロケータと

前記レジスタアロケータによるレジスタ割付の結果に基づいてオブジェクトコードを生成するコードジェネレータとを備え、

前記レジスタアロケータは、

前記プログラム中の命令に論理レジスタを割り当て、

前記プログラムにおけるプロシージャの呼び出し時に生存している物理レジスタがレジスタスタックの底から順に割り付けられるように、前記命令に割り当てられた論理レジスタを物理レジスタに割り当ててを特徴とするコンパイラ。

【請求項 9】 前記レジスタアロケータは、前記プログラムにおける重要度の高い部分に対して優先的に、前記論理レジスタの割り当て及び前記物理レジスタの割り当てを行い、

前記コードジェネレータは、前記プログラムにおける重要度の低い部分に対して、前記重要度の高い部分に対する前記論理レジスタの割り当て及び前記物理レジスタの割り当ての補償コードを生成することを特徴とする請求項 8 に記載のコンパイラ。

【請求項 10】 プログラミング言語で記述されたプログラムのソースコードを機械語コードに変換するコンパイラにおいて、

前記プログラム中の命令に対してレジスタを割り付けるレジスタアロケータと

前記レジスタアロケータによるレジスタ割付の結果に基づいてオブジェクトコードを生成するコードジェネレータとを備え、

前記コードジェネレータは、

レジスタが所定の命令に割り付けられているかどうかを確認しながらコード生成を行い、

プロシージャ呼び出し時に、計算ユニット資源に空きがある限りにおいて、レジスタスタック上で所定の命令に割り付けられているレジスタの内容を、命令に割り付けられておらず当該レジスタよりもレジスタスタックの底側に位置するレジスタにコピーすることを特徴とするコンパイラ。

【請求項 11】 プログラムのソースコードを入力する入力手段と、

入力されたソースコードをコンパイルして機械語コードに変換するコンパイラとを備え、

前記コンパイラは、前記プログラム中の所定のプロシージャにおいて他のプロシージャが呼び出される場合に、当該プロシージャ呼び出しに先立って、当該所定のプロシージャの実行のために割り付けられたレジスタのうちで当該他のプロ

シー ज्याの呼び出し時に生存しているレジスタを割り付け直すコードと、前記他のプロシー ज्याからの復帰が行われた場合に、レジスタイメージを割り付け直す前の状態に戻すコードとを生成することを特徴とするコンピュータ装置。

【請求項 1 2】 コンピュータを制御して実行プログラムを変換する変換プログラムであって、

前記実行プログラム中の命令に論理レジスタを割り当てる処理と、

前記実行プログラムにおけるプロシー ज्याの呼び出し時に生存している物理レジスタがレジスタスタックの底から順に割り付けられるように、前記論理レジスタと前記物理レジスタとの間のマッピングを行う処理と、

前記マッピングの結果に基づいてオブジェクトコードを生成する処理とを前記コンピュータに実行させることを特徴とする変換プログラム。

【請求項 1 3】 コンピュータを制御して実行プログラムを変換する変換プログラムであって、

レジスタが所定の命令に割り付けられているかどうかを確認しながらコード生成を行う処理と、

プロシー ज्या呼び出し時に、計算ユニット資源に空きがある限りにおいて、レジスタスタック上で所定の命令に割り付けられているレジスタの内容を、命令に割り付けられておらず当該レジスタよりもレジスタスタックの底側に位置するレジスタにコピーする処理とを前記コンピュータに実行させることを特徴とする変換プログラム。

【請求項 1 4】 コンピュータを制御して演算処理を実行するプログラムであって、

所定のプロシー ज्याにおいて他のプロシー ज्याが呼び出された場合に、当該所定のプロシー ज्याの実行のために割り付けられたレジスタのうちで当該他のプロシー ज्याの呼び出し時に生存しているレジスタを割り付け直して当該他のプロシー ज्याを呼び出す処理と、

前記他のプロシー ज्याからの復帰が行われた場合に、レジスタイメージを割り付け直す前の状態に戻す処理とを前記コンピュータに実行させることを特徴とするプログラム。

【請求項15】 コンピュータを制御して実行プログラムを変換する変換プログラムを格納した記憶媒体であって、

前記変換プログラムは、

前記実行プログラム中の命令に論理レジスタを割り当てる処理と、

前記実行プログラムにおけるプロシージャの呼び出し時に生存している物理レジスタがレジスタスタックの底から順に割り付けられるように、前記論理レジスタと前記物理レジスタとの間のマッピングを行う処理と、

前記マッピングの結果に基づいてオブジェクトコードを生成する処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【請求項16】 コンピュータを制御して実行プログラムを変換する変換プログラムを格納した記憶媒体であって、

前記変換プログラムは、

レジスタが所定の命令に割り付けられているかどうかを確認しながらコード生成を行う処理と、

プロシージャ呼び出し時に、計算ユニット資源に空きがある限りにおいて、レジスタスタック上で所定の命令に割り付けられているレジスタの内容を、命令に割り付けられておらず当該レジスタよりもレジスタスタックの底側に位置するレジスタにコピーする処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【請求項17】 コンピュータを制御して演算処理を実行するプログラムを格納した記憶媒体であって、

前記プログラムは、

所定のプロシージャにおいて他のプロシージャが呼び出された場合に、当該所定のプロシージャの実行のために割り付けられたレジスタのうちで当該他のプロシージャの呼び出し時に生存しているレジスタを割り付け直して当該他のプロシージャを呼び出す処理と、

前記他のプロシージャからの復帰が行われた場合に、レジスタイメージを割り付け直す前の状態に戻す処理とを前記コンピュータに実行させることを特徴とする記憶媒体。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】

本発明は、レジスタスタックを使用するCPUアーキテクチャに対してスタックレジスタを効率的に使用方法に関する。

【0002】

【従来の技術】

コンピュータにおける処理装置（CPU）のアーキテクチャには、レジスタをレジスタスタックとして使用するものがある。例えば、米国インテル社及び米国ヒューレット・パカード社による64ビットCPUアーキテクチャであるIA-64では、汎用レジスタの一部をレジスタスタックとしている。

【0003】

レジスタスタックでは、処理装置に用意されている物理レジスタの中からプロシージャ（procedure：手続き）が必要とする数のレジスタだけを論理レジスタとして確保して使用する（以下、説明において物理レジスタと論理レジスタとを区別する必要が無い場合は単にレジスタと表記する）。各プロシージャは、alloc命令により使用するレジスタの数を指定して割り当てを行い、呼び出し元に復帰するときにそれらのレジスタを開放する。ここで、各プロシージャは、alloc命令により指定された（以下、「割り付けられた」、「allocされた」などと表現する）数のレジスタを固定的に使用する。

レジスタスタック上のレジスタ（スタックレジスタ）は、ボトムからトップに向かい、in引数、ローカル変数、out引数として使用される。そして、プロシージャの呼び出し時にallocされたout引数が呼び出し先のプロシージャのin引数にリネームされる。

【0004】

これらのallocされるレジスタは、有限個の物理レジスタをリネームすることによって順番に確保される。そして、alloc時に物理レジスタが足りなくなった場合（スタックオーバーフロー）は、それまでallocされたレジスタを自動的にメモリに退避することによって、新しい物理レジスタを確保する。また、呼び出し

元への復帰時に、呼び出し元でallocされたレジスタがメモリに退避されている場合(スタックアンダーフロー)は、退避されていた値がレジスタに自動的に復元される。

このレジスタスタックを利用することにより、スタックオーバーフローやスタックアンダーフローが発生しない限り、レジスタの退避や復元にかかる処理コストを回避することができる。

【0005】

【発明が解決しようとする課題】

上述したように、CPUアーキテクチャにおいてレジスタスタックを利用すれば、スタックオーバーフローやスタックアンダーフローが発生しない限り、レジスタの退避や復元のコストを回避することができる。

ところで、最近では、命令の並列実行が可能なCPUの登場に伴い、プログラムのコンパイル技術において、命令の並列実行を意識したレジスタアロケーション手法が多々提案されている。これは、逆依存や出力依存の発生による並列実行可能性の阻害を削減するため、並列実行可能な命令に異なったレジスタを割り当てるという手法である。このため、プロシージャにおいて、allocされるレジスタ数は増える傾向にある。

【0006】

この手法は、処理装置の並列実行能力を引き出すために不可欠な手法であるが、一方で使用するレジスタ数の増大により、スタックオーバーフローやスタックアンダーフローの発生する頻度を高めてしまう。このスタックオーバーフローやスタックアンダーフローのコストは一般的に非常に高いため、過度のスタックオーバーフローやスタックアンダーフローの発生は、プログラムの実行性能を大きく損ねる原因となる。

【0007】

また、上述したように、従来、レジスタスタックを使用する場合、プロシージャは、alloc命令で指定した数のレジスタを、当該プロシージャの中で固定的に使用していた。

しかしながら、当該プロシージャから呼ばれる他のプロシージャの呼び出し時

に、allocされているレジスタが全て活用されているとは限らない。この場合、活用されていないレジスタをスタックに残したまま、新たなレジスタのallocが行われることとなり、スタック資源が浪費されていた。

【0008】

そこで、本発明は、レジスタスタックを使用するアーキテクチャを持った処理装置に対するプログラムコードの生成において、スタックオーバーフローやスタックアンダーフローの発生を抑制し、これらによるプログラムの実行性能の低下を防止することを目的とする。

【0009】

また、本発明は、レジスタスタックを使用するアーキテクチャを持った処理装置に対するプログラムコードの生成において、プロシージャの実行段階で活用されないレジスタが発生することを抑え、スタックレジスタを効率的に使用方法を提供することを他の目的とする。

【0010】

【課題を解決するための手段】

上記の目的を達成する本発明は、ソースコードで記述されたプログラムをオブジェクトコードに変換するコンパイル方法において、コンパイル対象であるプログラムに対してレジスタを割り付けるステップと、レジスタ割付の結果に基づいてオブジェクトコードを生成するステップとを含む。そして、レジスタを割り付けるステップは、プログラム中の命令に論理レジスタを割り当てるステップと、コンパイル対象のプログラムにおけるプロシージャの呼び出し時に生存している物理レジスタがレジスタスタックの底から順に割り付けられるように、論理レジスタと物理レジスタとの間のマッピングを行うステップとを含むことを特徴とする。

【0011】

ここで、このマッピングを行うステップにおいて、生存区間が重なるプロシージャ呼び出しが多い論理レジスタほど優先させてスタックの底側に割り当てられるように物理レジスタに割り付けるようにマッピングを行うことができる。

また、このマッピングを行うステップにおいて、生存区間が重なるプロシージ

ャ呼び出し時点において、同時に生きている論理レジスタの数が少ないほど、かかる論理レジスタを優先させてスタックの底側に割り当てられるように物理レジスタに割り付けるようにマッピングを行うことができる。

【 0 0 1 2 】

また、本発明は、コンピュータを制御するプログラムのコードを生成するコード生成方法において、レジスタが所定の命令に割り付けられているかどうかを確認しながらコード生成を行い、プロシージャ呼び出し時に、計算ユニット資源に空きがある限りにおいて、レジスタスタック上で所定の命令に割り付けられているレジスタの内容を、このレジスタよりもレジスタスタックの底側に位置し何らの命令にも割り付けられていないレジスタにコピーすることを特徴とする。

【 0 0 1 3 】

さらにまた、本発明は、レジスタスタックを使用するアーキテクチャを持った処理装置におけるプログラム実行時のスタックレジスタ使用方法において、所定のプロシージャにおいて他のプロシージャが呼び出された場合に、この所定のプロシージャの実行のために割り付けられたレジスタのうちでかかる他のプロシージャの呼び出し時に生存しているレジスタを割り付け直して、生存しているレジスタをスタック上に残し生存していないレジスタをスタックからできるだけ捨てた上で、かかる他のプロシージャを呼び出すステップと、かかる他のプロシージャからの復帰が行われた場合に、レジスタイメージを割り付け直す前の状態に戻すステップとを含むことを特徴とする。

【 0 0 1 4 】

ここで、レジスタを割り付け直して他のプロシージャを呼び出すステップは、かかる他のプロシージャの呼び出し時に生存しているレジスタを、レジスタスタックの底から順にソートして割り付け直すステップを含むことができる。

【 0 0 1 5 】

また、本発明は、レジスタスタックを使用するアーキテクチャを持った処理装置におけるプログラム実行時のスタックレジスタ使用方法において、プロシージャの呼び出しを行うたびに、このプロシージャの呼び出し時にスタックレジスタ上に生存しているレジスタをパッキングして割り付け、このプロシージャを実行

した後に、レジスタイメージをパッキング前の状態に戻すことを特徴とする。

【0016】

さらに本発明は、これらのコンパイル方法、コード生成方法及びスタックレジスタ使用方法のステップに対応する処理をコンピュータ装置に実行させるプログラムを提供することができる。このプログラムは、磁気ディスクや光ディスク、半導体メモリその他の記憶媒体に格納して配布したり、ネットワークに接続されたプログラム伝送装置の記憶装置に格納しこのネットワークを介して配信したりすることにより提供することができる。

【0017】

また、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語コードに変換するコンパイラにおいて、コンパイル対象であるプログラム中の命令に対してレジスタを割り付けるレジスタアロケータと、このレジスタアロケータによるレジスタ割付の結果に基づいてオブジェクトコードを生成するコードジェネレータとを備える。このレジスタアロケータは、コンパイル対象のプログラム中の命令に論理レジスタを割り当て、このプログラムにおけるプロシージャの呼び出し時に生存している物理レジスタがレジスタスタックの底から順に割り付けられるように、このプログラム中の命令に割り当てられた論理レジスタを物理レジスタに割り当てることを特徴とする。

【0018】

ここで、このレジスタアロケータは、コンパイル対象のプログラムにおける重要度の高い部分に対して優先的に、論理レジスタの割り当て及び物理レジスタの割り当てを行い、このコードジェネレータは、このプログラムにおける重要度の低い部分に対して、この重要度の高い部分に対する論理レジスタの割り当て及び物理レジスタの割り当ての補償コードを生成する構成とすることができる。

このプログラムの部分における重要度については、このプログラムを実行した場合における該当部分の実行頻度を参酌することができる。すなわち、実行頻度が高い部分を重要度の高い部分として優先的に論理レジスタの割り当て及び物理レジスタの割り当てを行う。

【0019】

さらに、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語コードに変換するコンパイラにおいて、コンパイル対象であるプログラム中の命令に対してレジスタを割り付けるレジスタアロケータと、このレジスタアロケータによるレジスタ割付の結果に基づいてオブジェクトコードを生成するコードジェネレータとを備える。このコードジェネレータは、レジスタが所定の命令に割り付けられているかどうかを確認しながらコード生成を行い、プロシージャ呼び出し時に、計算ユニット資源に空きがある限りにおいて、レジスタスタック上で所定の命令に割り付けられているレジスタの内容を、このレジスタよりもレジスタスタックの底側に位置し、何らの命令にも割り付けられていないレジスタにコピーすることを特徴とする。

【 0 0 2 0 】

さらに、本発明は、次のように構成されたことを特徴とするコンピュータ装置を提供することができる。すなわち、このコンピュータ装置は、プログラムのソースコードを入力する入力手段と、入力されたソースコードをコンパイルして機械語コードに変換するコンパイラとを備える。そして、このコンパイラは、コンパイル対象であるプログラム中の所定のプロシージャにおいて他のプロシージャが呼び出される場合に、このプロシージャ呼び出しに先立って、この所定のプロシージャの実行のために割り付けられたレジスタのうちでかかる他のプロシージャの呼び出し時に生存しているレジスタを、当該レジスタのみがスタック上に残るように割り付け直すコードと、かかる他のプロシージャからの復帰が行われた場合に、レジスタイメージを割り付け直す前の状態に戻すコードとを生成する。

【 0 0 2 1 】

【発明の実施の形態】

以下、添付図面に示す実施の形態に基づいて、この発明を詳細に説明する。

図 1 は、本実施の形態におけるコンパイラの全体構成を説明する図である。

図 1 に示すコンパイラ 10 は、入力手段 20 を介してコンパイル対象であるプログラム（ソースコード）を入力し、かかるプログラムに対してコンパイル処理を行って機械語で記述されたオブジェクトコードを生成し出力する。

図 1 を参照すると、本実施の形態のコンパイラ 10 は、コンパイル対象である

プログラム中の命令のレジスタ割付（レジスタアロケーション）を行うレジスタアロケータ 1 1 と、レジスタ割付が済んだソースコードをオブジェクトコードに変換するコードジェネレータ 1 2 とを備える。

【 0 0 2 2 】

レジスタアロケータ 1 1 は、カラーリングなどの手法により変数と論理レジスタとのマッピングを行い、さらにこの論理レジスタと物理レジスタとのマッピングを行うことにより、レジスタ割付を行う。本実施の形態では特に、レジスタスタックを効率よく使用するため、後述する手法を用いてレジスタマッピングを行う。

【 0 0 2 3 】

コードジェネレータ 1 2 は、コンパイル対象であるソースコードに対応するオブジェクトコードを生成する。この際、レジスタアロケータ 1 1 によるレジスタマッピングの結果に基づいて、実際にレジスタスタックを使用するためのalloc命令と引数の再セットを行う命令とを生成する。また、本実施の形態では、後述する手法を用いて、レジスタスタックを効率よく使用することを考慮したコード生成を行う。

【 0 0 2 4 】

図 1 に示したコンパイラ 1 0 は、パーソナルコンピュータやワークステーションその他のコンピュータ装置にて実現され、図 1 に示す各構成要素は、コンピュータプログラムにより制御されたCPUにて実現される仮想的なソフトウェアブロックである。CPUを制御する当該コンピュータプログラムは、CD-ROMやフロッピーディスクなどの記憶媒体に格納して配布したり、ネットワークを介して伝送したりすることにより提供される。

なお、図 1 に示したコンパイラ 1 0 の構成要素は、本実施の形態における特徴的な機能に関して記述したものである。図示しないが、実際にはコンパイラ 1 0 は、入力プログラムの字句解析や構文解析、最適化などのコンパイル処理における一般的な機能を有することは言うまでもない。

また、図 1 に示した入力手段 2 0 は、所定の記憶装置に格納されているプログラム（ソースコード）を読み出す読み取り装置や、ネットワークを介して他のコ

ンピュータ装置からプログラム（ソースコード）を受信するネットワークインターフェイスなどにより実現することができる。

【0025】

本実施の形態のコンパイラ10によりコンパイルされたオブジェクトコードは、IA-64のようなレジスタスタックを使用するCPUアーキテクチャを持ったCPUで動作することを前提としている。したがって、各プロシージャにおいて、alloc命令により使用するレジスタの数が指定され、指定された数のレジスタが物理レジスタから確保される。また、プロシージャの終了時に当該プロシージャにおいてallocされていたレジスタが開放される。

【0026】

さらに、本実施の形態では、並列指向のレジスタアロケータ11が多くのレジスタを必要としたとしても、初期的に確保されたレジスタの全てを、プロシージャ呼び出しを越えて保存する必要はないことに注目し、レジスタのallocを細密に実行する。

すなわち、プロシージャ中で他のプロシージャを呼び出す場合、allocされているレジスタのうちでプロシージャの呼び出し時に呼び出し元のプロシージャで活用され命令に割り当てられている（以下、この状態を生存している、生きている等と表現する）レジスタをスタックレジスタ上に再パッキングし、もう一度allocし直してからプロシージャを呼び出す。この課程で、プロシージャの呼び出し時に呼び出し元のプロシージャで活用されていない（以下、この状態を死んでいると表現する）レジスタが開放されるため、開放されるレジスタの数だけallocされているレジスタの数は減少する。

そして、呼び出されたプロシージャからの復帰が行われた時点で、レジスタイメージを元に戻す。これは、プロシージャ呼び出しを行った元のプロシージャにおいては、最大で、最初のalloc命令で確保した数のレジスタが必要だからである。言い換えれば、本実施の形態では、呼び出されたプロシージャを実行する間だけ、当該呼び出されたプロシージャで必要とするレジスタのみを確保して残りを開放する。このため、当該呼び出されたプロシージャから呼び出し元のプロシージャに復帰したならば、レジスタイメージを元に戻すことが必要となる。

このように、プロシージャの呼び出しごとにスタックレジスタをパッキングすることによって、各時点で死んでいるレジスタを有効活用することが可能となるため、スタックオーバーフローが発生する頻度を下げることができる。

【 0 0 2 7 】

しかしながら、プロシージャの呼び出しごとにレジスタのallocを細密に行っていくと、レジスタをパッキングする際に発生するレジスタ移動によって、実行時間がかえって増加してしまうことも考えられる。

そこで、本実施の形態では、レジスタ移動によるコード品質の低下を回避しながら、レジスタのパッキングを行う手法として、次の2つの手法を提案する。

第1の手法は、レジスタパッキングを考慮しながらレジスタアロケーションを行う手法である（後述の手法（1）（2））。これは、プロシージャの呼び出し時に生存しているレジスタを、予めスタックのボトムの方に割り付ける方法であり、図1のコンパイラ10におけるレジスタアロケータ11にて実行される。

第2の手法は、コード生成時にプロシージャ呼び出しごとにレジスタパッキングを行う手法である（後述の手法（3））。この手法は、図1のコンパイラ10におけるコードジェネレータ12にて実行される。

なお、レジスタアロケータ11にてレジスタパッキングを行う第1の手法の場合、コードジェネレータ12は、レジスタアロケータ11によるレジスタ割付の結果に基づいて通常のコード生成を行う。一方、コードジェネレータ12にてレジスタパッキングを行う第2の手法の場合、レジスタアロケータ11は、通常のレジスタ割付のみを行ってコードジェネレータ12に処理を渡す。

【 0 0 2 8 】

以下、各手法について詳細に説明する。

（1）レジスタパッキングを考慮しながらレジスタアロケーションを行う手法

この手法は、グラフカラーリング等の手法により変数と論理レジスタとのマッピングが決定された後、プロシージャ呼び出し時におけるレジスタパッキングを考慮しながら、論理レジスタと物理レジスタとのマッピングを決定する方法である。

本手法では、以下の2つの戦略に基づいて論理レジスタの優先順位を決め、こ

の順位にしたがってレジスタスタックの底（ボトム）の方から物理レジスタを割り当てていく。

戦略1：生存区間が重なるプロシージャ呼び出しが多い論理レジスタほど優先させる。

戦略2：生存区間が重なるプロシージャ呼び出し時点において、同時に生きている論理レジスタ数が少ないほど優先させる。

【0029】

上記2つの戦略を、図2乃至図4を参照して説明する。

図2は、9個のレジスタ（a～i）をallocしたプロシージャにおいて、3つのプロシージャ呼び出し（図のcall[1]、call[2]、call[3]）があったときの、論理レジスタa～iの生存区間を示す図であり、図3は上述した戦略1に基づいて図2の論理レジスタをソートした様子を示す図、図4はさらに上述した戦略2に基づいて図3の論理レジスタをソートした様子を示す図である。

図2に示すように、a～iの順番でスタックの底からレジスタを割り付けた場合、call[1]のプロシージャ呼び出し時には3つのレジスタf、g、hが死んでいる。同様にcall[2]のプロシージャ呼び出し時には5つ（a、d、e、g、h）、call[3]のプロシージャ呼び出し時には2つ（c、h）のレジスタが死んでいる。したがって、これらのレジスタが無駄となっている。

【0030】

そこで、戦略1を用い、生存区間が重なるプロシージャ呼び出しが多い論理レジスタほど優先的にレジスタスタックの底に割り付けることを考える。図3に、このようにしてソートした様子を示す。なお、図3において、生存区間が重なるプロシージャ呼び出しが同じ論理レジスタについては辞書的順序で並べてある。

図3を参照すると、3つのプロシージャ呼び出し全てにまたがる論理レジスタb、iがレジスタスタックの底に移動している。このため、call[1]で最上位の3つのレジスタが空いており、同様にcall[2]では2つ、call[3]では1つのレジスタが空いている。したがって、各プロシージャ呼び出しの呼び出し先であるプロシージャの実行時には、これらのレジスタを開放して再allocが可能である。

しかし、この状態でも、call[2]のプロシージャ呼び出し時には3つのレジス

タ a、d、e が、call [3] のプロシージャ呼び出し時には 1 つのレジスタ f、g、h がそれぞれ死んでおり、無駄になっている。

【0031】

そこで、さらに戦略 2 を用い、生存区間が重なるプロシージャ呼び出し時点において、同時に生きている論理レジスタ数が少ないほど優先させてレジスタスタックの底に割り付けることを考える。図 2 及び図 3 の場合、生存区間が重なるプロシージャ呼び出しの数が 2 つであるレジスタ a、c、d、e、f について考えると、call [2] の呼び出し時に生きている論理レジスタ数は 2 つであり、他の呼び出し時（いずれも 4 つ）と比べて少ない。したがって、この時点で干渉を持っている論理レジスタ c や論理レジスタ f の優先度を高くする。図 4 に、このようにしてソートした様子を示す。なお、図 4 において、生存区間が重なるプロシージャ呼び出しが同じ論理レジスタについては辞書的順序で並べてある。

図 4 を参照すると、上述したように、論理レジスタ c、f がボトム方向へ移動し、論理レジスタ b、i の次に置かれている。これにより、call [1] の最上位で空いているレジスタ数は 1 つ減って 2 つとなっているが、call [2] では 5 つのレジスタが空いている。また、call [3] では 1 つのレジスタが空いている。したがって、各プロシージャ呼び出しの呼び出し先であるプロシージャの実行時には、これらのレジスタを開放して再allocが可能である。

【0032】

以上の処理により、最終的に、call [1]、call [2]、call [3] の全体で、死んだまま残るレジスタ数は 2 つとなり、図 2 の状態と比べると 8 つのレジスタが有効活用されており、最適な結果が得られていることがわかる。

なお、戦略 2 の方法によるソートは、戦略 1 の方法によるソートよりも重み付けを低くする。したがって、戦略 2 の方法によるソートでは、戦略 1 の方法により（プロシージャ呼び出しの多さで）決定された順序を覆すことはない。例えば、論理レジスタ g が仮に call [3] ではなく call [2] で生きていたとしても、図 4 の状態で論理レジスタ e よりもボトム側へ移動することはない。

【0033】

図 5 は、レジスタアロケータ 11 による本手法を用いたレジスタ割付のアルゴ

リズムを説明するフローチャートである。

図5に示すように、レジスタアロケータ11は、まず、変数の生存区間を解析する（ステップ501）。そして、この解析結果に基づき、論理レジスタを対象として、通常のレジスタ割付を行う（ステップ502）。

【0034】

次に、レジスタアロケータ11は、各論理レジスタについて、その生存区間内に存在するプロシージャ呼び出しの数をカウントする（ステップ503）。また、各プロシージャ呼び出しについて、生存区間が重なっている論理レジスタの数をカウントする（ステップ504）。このステップ503、504はいずれを先に行ってもかまわない。

この後、レジスタアロケータ11は、論理レジスタを、生存区間が重なるプロシージャ呼び出しの数が多い順にソートする（ステップ505）。なお、プロシージャ呼び出しの数と同じ場合は、例えば辞書的順序にしたがってソートする。

次に、レジスタアロケータ11は、生存区間が重なるプロシージャ呼び出しの数と同じ論理レジスタについて、プロシージャ呼び出し時に同時に生きている論理レジスタ数が少ないものから順にソートする（ステップ506）。具体的には、所定の論理レジスタに関して、当該論理レジスタと生存区間が重なっているプロシージャ呼び出しを全て検出し、それらのプロシージャ呼び出しと生存区間が重なっている論理レジスタの数について平均値を取る。なお、プロシージャ呼び出し時に同時に生きている論理レジスタ数が同じ場合は、例えば辞書的順序にしたがってソートする。

【0035】

この後、レジスタアロケータ11は、ステップ505及びステップ506で行ったソートの結果に基づき、物理レジスタであるスタックレジスタを、レジスタスタックのボトムの方から順に、論理レジスタにマッピングする（ステップ507）。

【0036】

以上のようにして、レジスタアロケータ11によるレジスタ割付が済んだ後、コードジェネレータ12により実際にスタックレジスタの効率的な使用を実現す

るコードが生成される。

図 6 は、コードジェネレータ 1 2 がレジスタ割付の結果を利用してスタックレジスタを再allocする動作を説明するフローチャートである。なお、コードジェネレータ 1 2 は、図 6 に示す処理を行う前に、通常のコード生成をすでに行っており、かかるコード生成処理の後、プログラム中の各プロシージャに対して、図 6 に示す処理を行う。

【 0 0 3 7 】

図 6 を参照すると、コードジェネレータ 1 2 は、まず、着目したプロシージャ呼び出し時に生きている論理レジスタのうちでレジスタスタックの最もトップ側にある論理レジスタを検出する（ステップ 6 0 1、6 0 2）。そして、検出された論理レジスタのレジスタスタック上の位置を、再allocされるレジスタスタックの新しいスタックトップであるとして、新しいフレームサイズを計算する（ステップ 6 0 3）。

次に、コードジェネレータ 1 2 は、ステップ 6 0 2 で算出された新しいフレームサイズに基づいて、引数の積み直しを行う（ステップ 6 0 4）。具体的には、まず、新しい引数の積み先が空いているものについて引数を積む。そしてその結果、引数の積み先が空いたものに関してさらに引数を積む。この操作を空いた引数の積み先が無くなるまで繰り返す。また、未処理の引数については、ワーキングレジスタ等を使いながら、引数を積む。

【 0 0 3 8 】

以上のようにして引数の積み直しが済んだならば、次に、新しいフレームを生成するalloc命令、プロシージャ呼び出し命令（call命令）、フレームを元の状態に戻すためのalloc命令を生成する（ステップ 6 0 5）。

ステップ 6 0 2 ～ 6 0 5 の各処理を、プログラム中の全てのプロシージャに対して行ったならば、処理を終了する（ステップ 6 0 1）。

【 0 0 3 9 】

（ 2 ） プログラムの局所性を利用した手法

この手法は、（ 1 ）の手法を基本としながら、その手法をプログラムに対して局所的に適用する方法である。

プログラムは、通常、複数の部分（パート）から構成されており、部分によって重要性が異なる場合がある。そこで、本手法では、（１）の手法のようにプログラム全体に一括して適用するのではなく、プログラムの重要な部分を優先してレジスタパッキングを行う。

すなわち、プロファイル情報等を利用して重要度の高いホットスポットとなるパスを特定し、当該パスに存在するプロシージャ呼び出し時にレジスタが最大限にパッキングされるようにレジスタ割付を行う。そして、ホットスポットとならないパスには、ホットスポットとなるパスに対するパッキングのための補償コードを出す。

なお、プログラムの部分における重要度の高低は、相対的に決まる。すなわち、最も重要度の高い部分において最も効率的にスタックレジスタを使用できるようにパッキングが行われ、部分の重要度が低くなるにつれて、より重要度の高い部分のパッキングに対する補償コードが生成されることとなる。

【 0 0 4 0 】

図 7 は、レジスタアロケータ 1 1 による本手法を用いたレジスタ割付のアルゴリズムを説明するフローチャートである。本手法は、レジスタ移動がホットスポットに生成されないことを考慮し、図 7 のフローチャートに示すアルゴリズムで実現される。

図 7 に示すように、コンパイラ 1 0 のレジスタアロケータ 1 1 は、まず、プログラムをトレースに分割する（ステップ 7 0 1）。そして、各トレースを、実行頻度等の重要度に基づいてソートする（ステップ 7 0 2）。

次に、レジスタアロケータ 1 1 は、レジスタ移動のパッドを作るため、必要に応じて重要度の低いトレースを分割する（ステップ 7 0 3）。具体的には、重要度の低いトレースの途中から、重要度の高いトレースへの流出がある場合、その点で重要度の低いトレースを分割する。また、重要度の低いトレースの途中に、重要度の高いトレースからの流入がある場合、その点で重要度の低いトレースを分割する。

【 0 0 4 1 】

次に、レジスタアロケータ 1 1 は、上記のようにソートされた各トレースを順

次着目して（すなわち重要度の高い順に）、以下の処理を行う（ステップ704、705）。

まず、着目したトレースを対象として、（1）の手法による論理レジスタのソートを実行する（ステップ706）。

次に、着目中のトレースの先頭に他のトレースにおける論理レジスタと物理レジスタのマッピングが伝播されているかどうかを調べ、伝播されている場合、着目中のトレースの先頭に、伝播されているマッピングとステップ706によるソートの結果との擦り合わせを行うためのレジスタ移動を生成する（ステップ707、708）。

【0042】

また、着目したトレースから所定のトレースに対する流出があるかどうかを調べ、流出がない場合は処理を終了する（ステップ709）。一方、流出がある場合、その流出先のトレースにおいて先頭の論理レジスタと物理レジスタとのマッピングが決定されているかどうかをさらに調べる（ステップ709、710）。

そして、流出先のトレースにおける先頭の論理レジスタと物理レジスタとのマッピングが未決定である場合、レジスタアロケータ11は、着目中のトレースにおける論理レジスタと物理レジスタのマッピングを流出先のトレースの先頭に伝播させる（ステップ710、711）。ここで、重要度が高い順にトレースに着目していることから、流出先のトレースの重要度が流出元である着目中のトレースよりも高くないことが保証されている。

【0043】

一方、流出先のトレースにおける先頭の論理レジスタと物理レジスタとのマッピングが決定済みである場合、流出先のトレースの方が流出元である着目中のトレースよりも重要度が高いことがわかる。ここで、流出先のトレースにおける先頭の論理レジスタと物理レジスタとのマッピングが決定済みとなり得るのは、着目中のトレースの末尾からの流出先であるトレースに限られる。なぜなら、ステップ703において、重要度の低いトレースの途中から重要度の高いトレースへの流出がある場合に、当該重要度の低いトレースを流出点で分割していることから、トレースの途中からの流出先であるトレースにおいて先頭の論理レジスタと

物理レジスタのマッピングは未決定であることが保証されているためである。

したがって、着目中のトレースの末尾に、着目中のトレースに対するステップ 706 によるソートの結果と流出先のトレースにおける論理レジスタと物理レジスタのマッピングとの擦り合わせを行うためのレジスタ移動を生成する（ステップ 710、712）。

【0044】

ステップ 706 乃至ステップ 713 に示す処理を全てのトレースに対して行ったらば、レジスタアロケータ 11 によるレジスタ割付を終了する（ステップ 704）。

次に、コードジェネレータ 12 により、レジスタアロケータ 11 によるレジスタ割付の結果に基づいて、実際にスタックレジスタの効率的な使用を実現するコードが生成される。コードジェネレータ 12 によるコード生成処理の詳細は、図 6 のフローチャートに示した（1）の手法における処理と同様であるため、説明を省略する。

【0045】

以上の処理により、プログラム中の重要度の高い部分（実行頻度の高い部分など）に対して、優先的にスタックレジスタの効率的な使用を行うことが可能となる。したがって、プログラムの部分に着目すれば、スタックレジスタの使用効率が高くない部分が生じ得るが、プログラム全体に着目すれば、実行性能をより一層向上させることができる。

【0046】

なお、上述した（1）（2）の手法による処理は、レジスタアロケータ 11 によるレジスタ割付処理の一部として実行される。このように、レジスタパッキングを考慮したアロケーションの結果にしたがって、次にコードジェネレータ 12 によるコード生成が行われるため、レジスタをパッキングする際に発生するレジスタ移動によってプログラムの実行時間が増加してしまうことはない。

【0047】

（3）コード生成時にプロシージャ呼び出しごとにレジスタパッキングを行う手法

この手法は、レジスタが生存しているかどうかをトラックしながらコード生成を行い、プロシージャ呼び出し時に、計算ユニット資源に空きがある限りにおいて、スタックのトップの方に存在する生きているレジスタの内容をスタックのボトムの方に存在する死んでいるレジスタにコピーする手法である。

命令の並列実行が可能な処理装置では、一度に複数の命令を実行することができるが、命令間における同期の必要のため、常に処理装置の実行能力（計算ユニット資源）に見合った数の命令が実行されているわけではない。そこで、本手法では、この計算ユニット資源に空きがある場合に、当該空き資源を用いてレジスタパッキングを行う。

【0048】

図8は、コードジェネレータ12による本手法を用いたコード生成のアルゴリズムを説明するフローチャートである。本手法は、図8のフローチャートに示すアルゴリズムで実現される。

図8に示すように、コンパイラ10のコードジェネレータ12は、まず、プロシージャ呼び出しを検出したときのインストラクションスケジューリングウィンドウ（以下、単にウィンドウと記す。図において同じ）のフラッシュ時にウィンドウ内を検索し、alloc命令が生成できるかどうか及びそのときにコピー命令を何個生成できるかを検出する（ステップ801）。

次に、コードジェネレータ12は、当該プロシージャ呼び出し時に死んでいるスタックレジスタを検出し、スタックのボトム側にあるものから順にソートする（ステップ802）。

次に、コードジェネレータ12は、ウィンドウ内でセットされていない引数を検出し、そのような引数を再セットするためのコピー命令スロットを確保する（ステップ803）。ここで、コピー命令スロットを確保できなければ、本手法による処理を終了する（ステップ804）。

次に、コードジェネレータ12は、当該プロシージャ呼び出し時に生きているスタックレジスタを検出し、スタックのトップ側にあるものから順にソートする（ステップ805）。そして、ソートされたスタックレジスタを順番に、コピー命令スロットが確保される限り、死んでいるスタックレジスタにコピーする（ス

トップ806)。ここで、死んでいるスタックレジスタは、スタックのボトム側から順に使用する。

次に、コードジェネレータ12は、確保されたalloc命令の生成位置に、レジスタスタックを縮小したalloc命令を生成する(ステップ807)。そして、縮小結果に合わせて引数のリネームを行う(ステップ808)。ウィンドウ内でセットされていない引数については、ステップ803で確保された場所に、引数を再セットするためのコピー命令を生成する(ステップ809)。

最後に、コードジェネレータ12は、プロシージャ呼び出しの後ろに、上記のように移動したレジスタイメージを元に戻すためのコピー命令とalloc命令を生成する(ステップ810)。

【0049】

以上の処理により、処理装置の計算ユニット資源に空きがあり、レジスタパッキングを実行する余力がある場合にのみ、レジスタの内容をレジスタスタックのボトム方向へ移動し、トップ側のレジスタを開放できるようにする。したがって、レジスタパッキングによりスタックレジスタの効率的な使用を実現する一方で、レジスタをパッキングする際に発生するレジスタ移動によってプログラムの実行時間が増加することを確実に防止することができる。

【0050】

【発明の効果】

以上説明したように、本発明によれば、レジスタスタックを使用するアーキテクチャを持った処理装置に対するプログラムコードの生成において、スタックオーバーフローやスタックアンダーフローの発生を抑制し、これらによるプログラムの実行性能の低下を防止することが可能となる。

【0051】

また、本発明によれば、かかるプログラムコードの生成において、プロシージャの実行段階で活用されないレジスタが発生することを抑え、スタックレジスタを効率的に使用することが可能となる。

【図面の簡単な説明】

【図1】 本実施の形態におけるコンパイラの全体構成を説明する図。

【図 2】 本実施の形態において、9 個のレジスタを alloc したプロシージャで 3 つのプロシージャ呼び出しがあったときの各論理レジスタの生存区間を示す図である。

【図 3】 図 2 の状態に対し本実施の形態の手法に基づいて論理レジスタをソートした様子を示す図である。

【図 4】 図 3 の状態に対し本実施の形態の他の手法に基づいて論理レジスタをソートした様子を示す図である。

【図 5】 本実施の形態のレジスタアロケータによるレジスタ割付のアルゴリズムを説明するフローチャートである。

【図 6】 本実施の形態のコードジェネレータがレジスタアロケータによるレジスタ割付の結果を利用してスタックレジスタを再 alloc する動作を説明するフローチャートである。

【図 7】 本実施の形態のレジスタアロケータによるレジスタ割付の他のアルゴリズムを説明するフローチャートである。

【図 8】 本実施の形態のコードジェネレータによるコード生成のアルゴリズムを説明するフローチャートである。

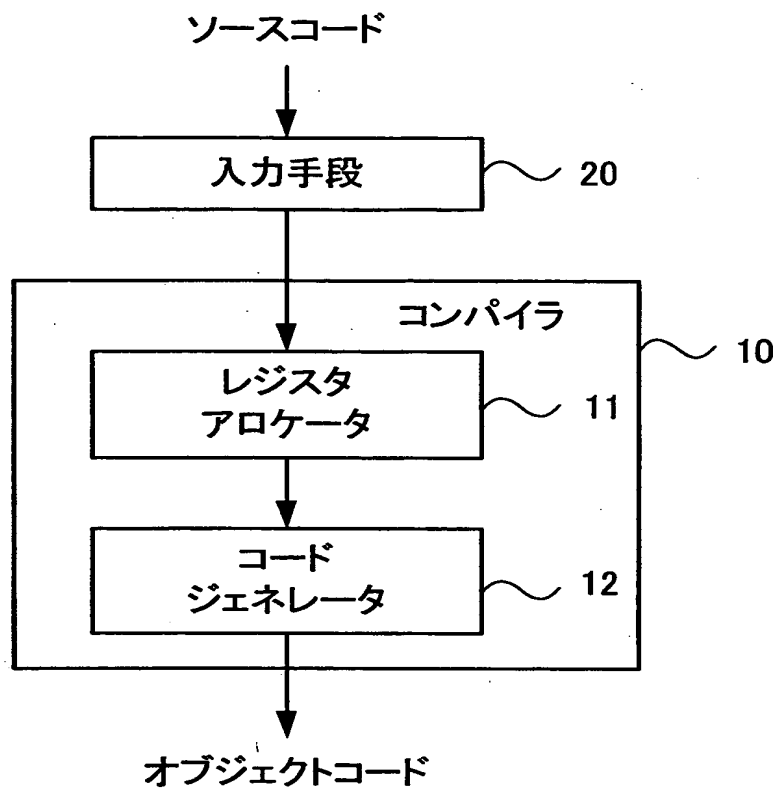
【符号の説明】

1 0 …コンパイラ、1 1 …レジスタアロケータ、1 2 …コードジェネレータ、2 0 …入力手段

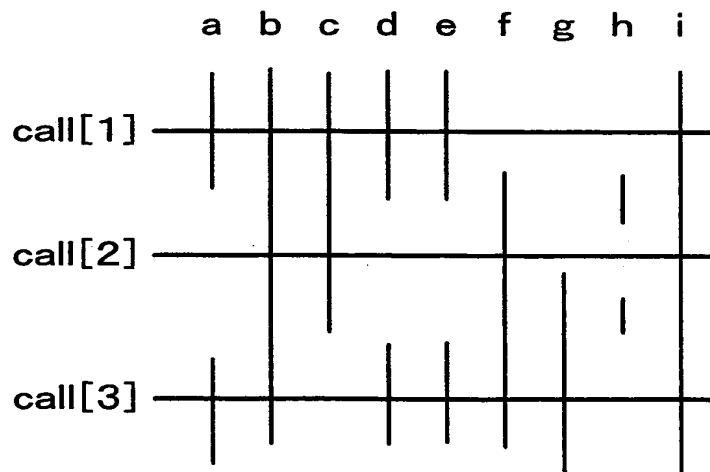
【書類名】

図面

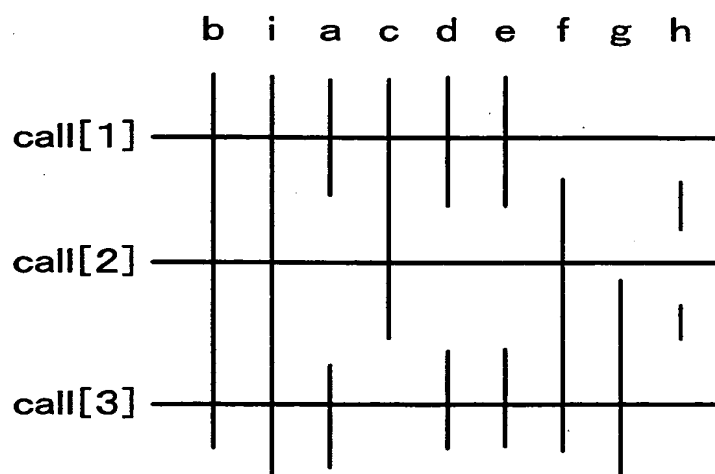
【図 1】



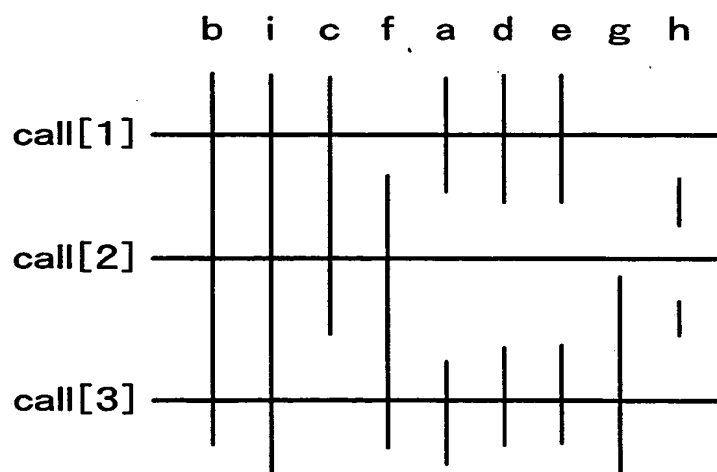
【図 2】



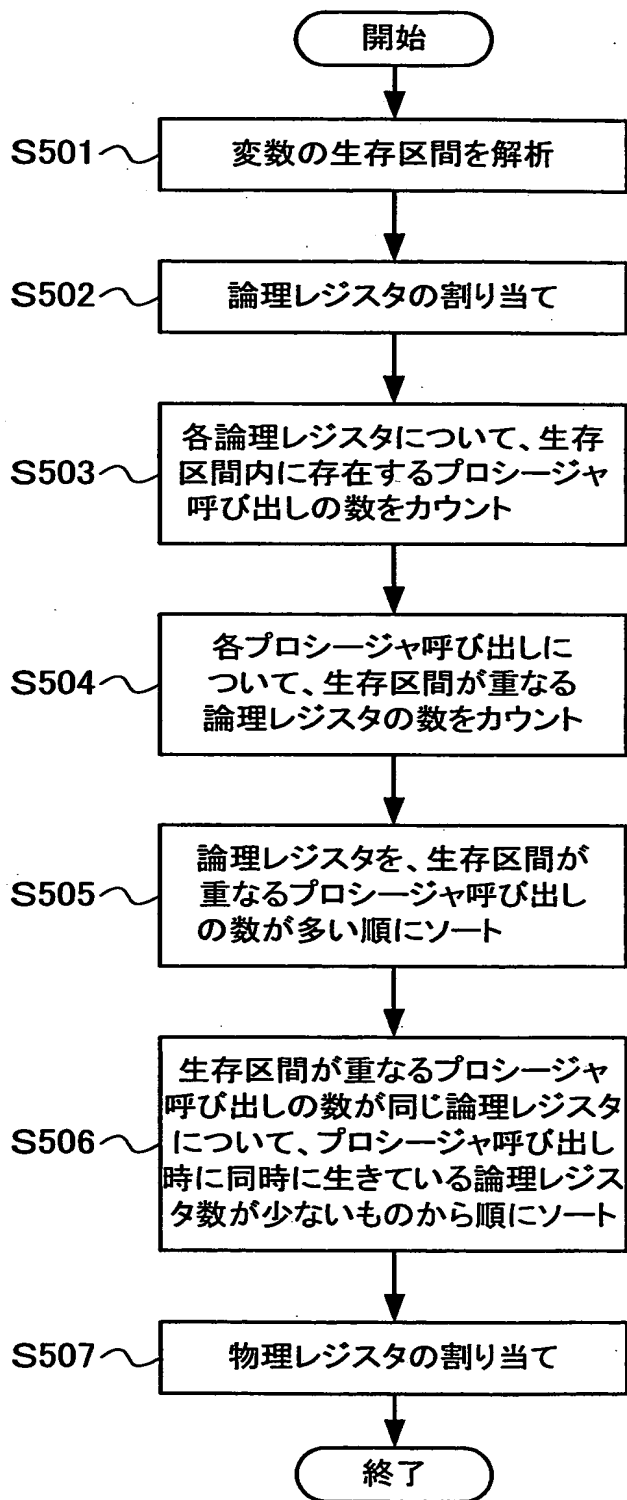
【図3】



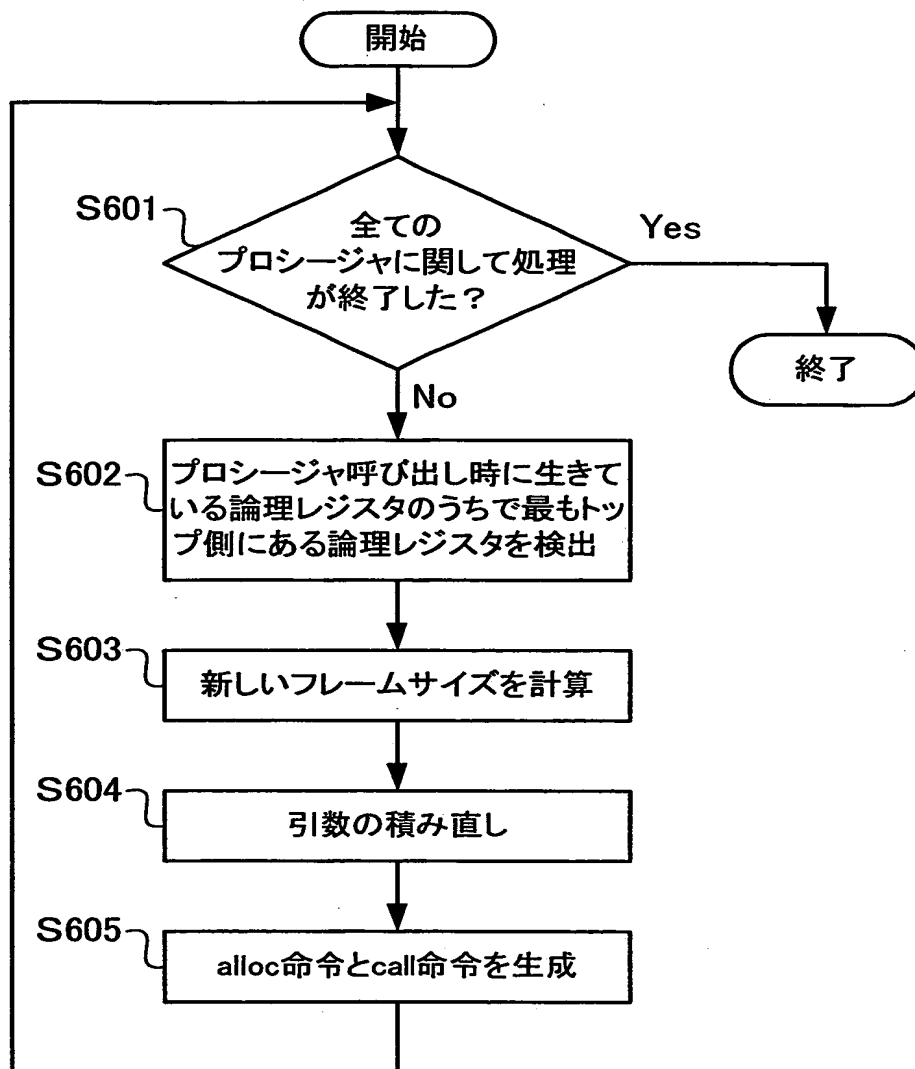
【図4】



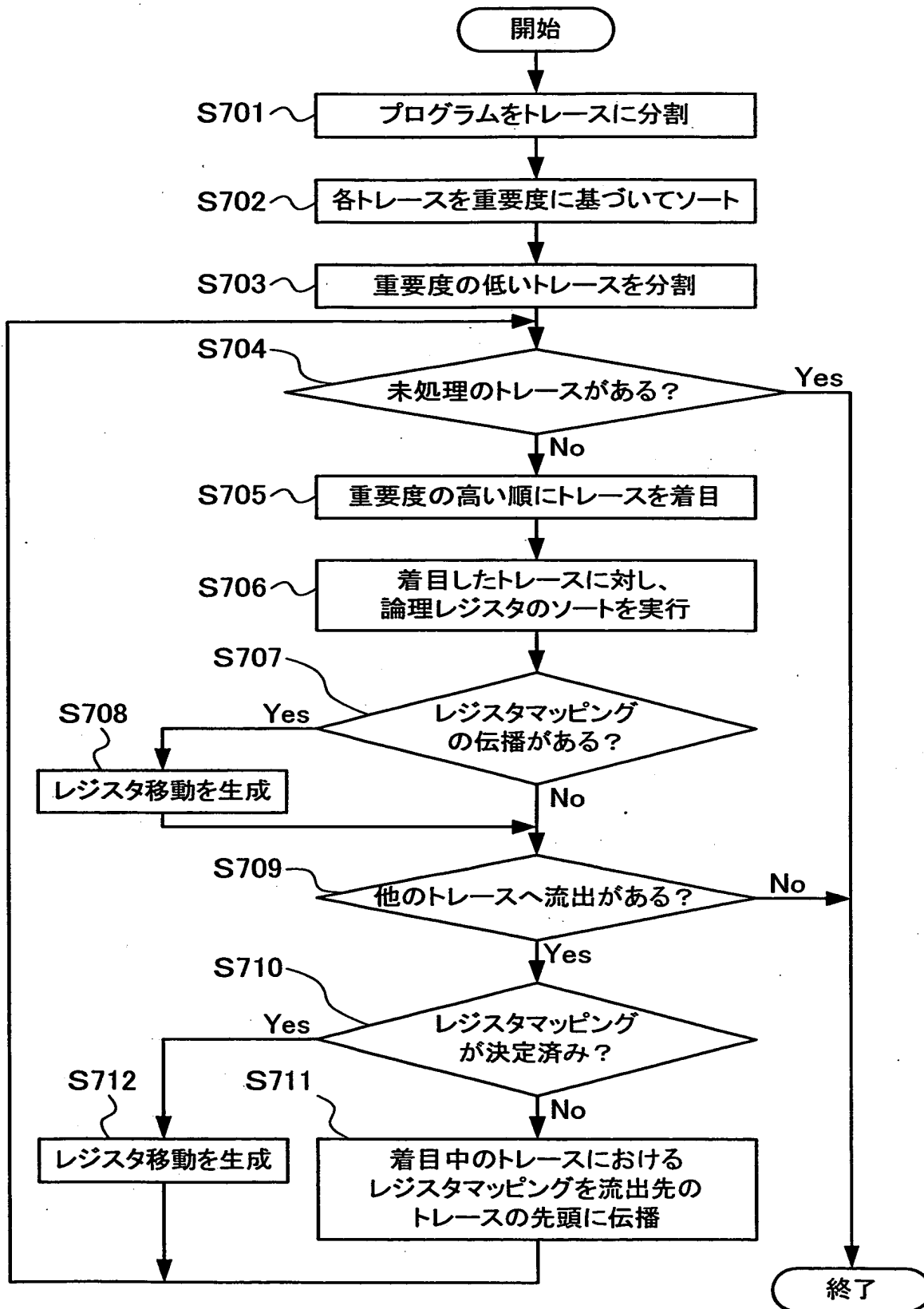
【図 5】



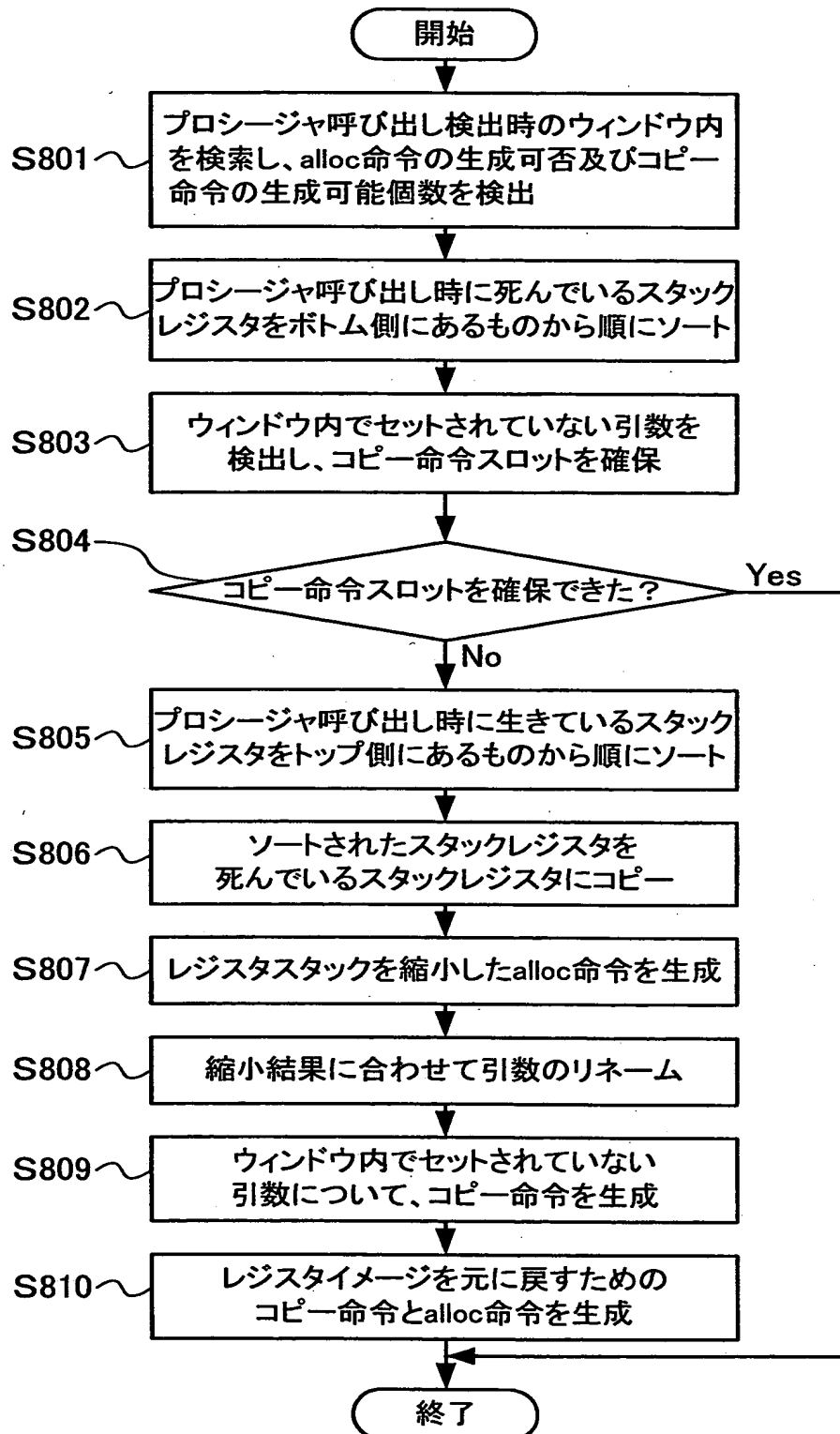
【図 6】



【図 7】



【図 8】



【書類名】 要約書

【要約】

【課題】 レジスタスタックを使用するアーキテクチャを持った処理装置に対するプログラムコードの生成において、スタックオーバーフローやスタックアンダーフローの発生を抑制し、これらによるプログラムの実行性能の低下を防止する。

【解決手段】 コンパイラ 1 0 において、コンパイル対象であるプログラム中の命令に対してレジスタを割り付けるレジスタアロケータ 1 1 と、このレジスタアロケータ 1 1 によるレジスタ割付の結果に基づいてオブジェクトコードを生成するコードジェネレータ 1 2 とを備える。このレジスタアロケータ 1 1 は、コンパイル対象のプログラム中の命令に論理レジスタを割り当て、コンパイル対象のプログラムにおけるプロシージャの呼び出し時に生存している物理レジスタがレジスタスタックの底から順に割り付けられるように、このプログラム中の命令に割り当てられた論理レジスタを物理レジスタに割り当てる。

【選択図】 図 1

認定・付加情報

特許出願の番号	特願2001-161364
受付番号	50100773239
書類名	特許願
担当官	風戸 勝利 9083
作成日	平成13年 7月10日

<認定情報・付加情報>

【特許出願人】

【識別番号】	390009531
【住所又は居所】	アメリカ合衆国10504、ニューヨーク州 アーモンク (番地なし)
【氏名又は名称】	インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

【識別番号】	100086243
【住所又は居所】	神奈川県大和市下鶴間1623番地14 日本アイ・ビー・エム株式会社 大和事業所内
【氏名又は名称】	坂口 博

【代理人】

【識別番号】	100091568
【住所又は居所】	神奈川県大和市下鶴間1623番地14 日本アイ・ビー・エム株式会社 大和事業所内
【氏名又は名称】	市位 嘉宏

【代理人】

【識別番号】	100106699
【住所又は居所】	神奈川県大和市下鶴間1623番14 日本アイ・ビー・エム株式会社大和事業所内
【氏名又は名称】	渡部 弘道

【復代理人】

【識別番号】	100104880
【住所又は居所】	東京都港区赤坂5-4-11 山口建設第2ビル 6F セリオ国際特許事務所
【氏名又は名称】	古部 次郎

【選任した復代理人】

【識別番号】	100100077
--------	-----------

認定・付加情報（続き）

【住所又は居所】 東京都港区赤坂5-4-11 山口建設第2ビル
6F セリオ国際特許事務所
【氏名又は名称】 大場 充

出 願 人 履 歴 情 報

識別番号 [390009531]

1. 変更年月日 2000年 5月16日
[変更理由] 名称変更
住 所 アメリカ合衆国10504、ニューヨーク州 アーモンク (番地なし)
氏 名 インターナショナル・ビジネス・マシーンズ・コーポレーション